# Malwasm: A Detection and Analysis Tool for Wasm Binaries in Websites

Xavier Lim Gui Ming[#1], Chow Wen Jun[#2], Lim Yang Jun[#3], Chong Wei Bing Nicholas[#4]

[#]*Information and Communication Technology, Singapore Institute of Technology*

*Nanyang Polytechnic, 172A Ang Mo Kio Avenue 8, Singapore 567739*

[1]2000952@sit.singaporetech.edu.sg

[2]2000530@sit.singaporetech.edu.sg

[3]2000572@sit.singaporetech.edu.sg

[4]2001898@sit.singaporetech.edu.sg

***Abstract* - As WebAssembly (abbreviated Wasm) comes into prominence, malware developers are searching for methods to utilize it to serve malicious WebAssembly programs. The malicious code is coded in common programming languages such as C and Rust before it is compiled into a .wasm file extension. The wasm binary file is subsequently served on a web page and will be executed whenever a user browses the web page. This paper presents the implementation of Malwasm, a command-line tool with the capability to detect and extract wasm binaries used in websites and to perform analysis on the wasm files to determine if it contains any malicious functions.**

***Keywords*** - WebAssembly, Wasm, Malware Detection, Malware Analysis

## I. Introduction

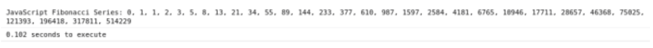
The ubiquity of the web has made web applications a common target for malicious actors. As a result, various attack methods such as browser-based crypto mining without user consent [1] and malicious malware code executed via scripts have risen to target the web browser. In the area of WebAssembly, a study conducted by cybersecurity company CrowdStrike has shown that 75% of wasm modules used in websites are malicious [8].

For every malware that is known, there would be a devised approach to detecting it. Typically, there are two known methodologies to detect malware [2], either using signature-based detection or heuristic-based detection. During the course of this research, it was important in identifying the current state of existing tools or solutions with the capability to detect and analyze malicious functions hidden within wasm binary files on websites. The result of the research has shown that there does not yet exist such a solution that is capable of doing both detection and analysis of wasm files. Most commercial AntiVirus software is able to detect malicious wasm files through a blacklist of keywords [10] but can be easily bypassed using obfuscation techniques. Thus, this has prompted the development of Malwasm, a command-line tool capable of detecting and analyzing potential malicious behavior in wasm files.

## II. Background Research

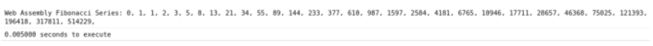
For a long period, JavaScript has been the sole way to create interactive applications in the browser. However, its poor performance due to inefficiencies has hampered the development of CPU-intensive applications like games on browsers. This has resulted in the introduction of WebAssembly [3] in 2015, a new low-level bytecode language that can be compiled within high-level languages such as C and Rust to output wasm code. By late 2017, WebAssembly is supported on all major browsers such as Microsoft Edge, Google Chrome, Mozilla Firefox, etc. Being a low-level language allows the wasm code to execute at near-native speed while running in a memory-safe, sandboxed environment in the browser [7].

WebAssembly is meant to address the slow load times [3] that JavaScript offers by compiling the lightweight C or Rust codes to a WebAssembly format, offering significant performance improvements. A visible performance difference could be seen in the example shown below when running the Fibonacci sequence function.



```
JavaScript Fibonacci Series: 0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, 144, 233, 377, 610, 987, 1597, 2584, 4181, 6765, 10946, 17711, 28657, 46368, 75025,
121393, 196418, 317811, 514229
0.102 seconds to execute
```

Figure 1. JavaScript Fibonacci Sequence



```
Web Assembly Fibonacci Series: 0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, 144, 233, 377, 610, 987, 1597, 2584, 4181, 6765, 10946, 17711, 28657, 46368, 75025, 121393,
196418, 317811, 514229,
0.005000 seconds to execute
```

Figure 2. WebAssembly Fibonacci Sequence

The WebAssembly based malware, cryptonight malware, is compiled to a wasm file that utilizes the victim's host computing resources to perform mining for the cryptocurrency Bitcoin. In this paper, the malware will be used to carry out the research and execution of the proposed tool.

## III. Technical Solution

The proposed technical solution would be Malwasm. Malwasm is a command-line utility tool, designed to extract wasm files from websites and disassemble the wasm binaries to generate rules based on semantic profiling [4]. Semantic profiling will analyze each function's instruction distribution as opposed to searching for strings in a wasm binary. The tool can perform a comprehensive analysis based on the generated rules to detect the presence of malicious functions in the files. Apart from a detection tool, it has capabilities to provide 3 different types of graphs to the user. The 3 graphs are the call graph, the control flow graph, and the data flow graph. The tool also includes a string-based detection based on YARA rules.

### *A. System Architecture*

The system architecture diagram is as shown in Figure 3 below. External resources such as the yara rule files and wasp.exe are present in the resources folder. Firstly, users supply a total of 2 inputs into Malwasm. The first input would be a specific URL that is necessary for the extraction of wasm files. The wasm files will be utilized for 3 main purposes i.e., security analysis, disassembly, and generation of graphs. The disassembly function will output 2 files, a dis.txt and a WebAssembly text file which are useful to users for manual analysis of the wasm file. On top of that, a JSON rule file pertaining to a particular malware can be generated from Malwasm to be subsequently used for malware detection purposes. The malware detection result will be presented on the command line output and output into an analysis.txt file as well. Yara functions in Malwasm are used to analyze the wasm file with the yara rules present in the resources/yara_rules directory. The results are shown in the command line upon executing the program. In addition, the 2nd input would be the specific function name. The function name is used for the generation of data flow graphs and control flow graphs.

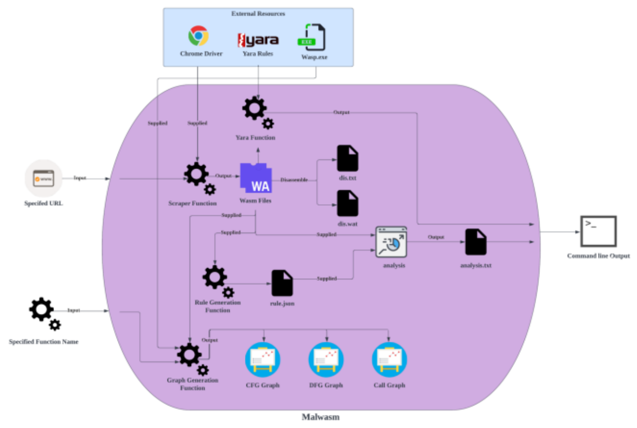


Figure 3. System Architecture Diagram

## *B. Technologies Involved*

In this section, the technologies that were used in this solution will be discussed in detail. The program developed by the team is purely developed with the python programming language. However, the program involves the usage of two external resources known as wasp.exe and the yara rule files. Table I below indicates and explains the various important python libraries that the solution relies on.

TABLE I
PYTHON LIBRARIES USED

| **Python Libraries** | **Usage** |
|---|---|
| argparse | This python library is used for the creation of user-friendly arguments for the program. |
| subprocess | This python library is used for the execution of executables present in the resource folder. |
| os | This python library is used for retrieving various directory paths. |
| json | This python library is used to parse JSON data to a python dictionary. |
| graphviz | This python library is used to render the dotfiles to SVG files. |
| glob | This python library is used to retrieve path names based on specified patterns. |
| yara-python | This python library is used to utilize YARA rules for compilation and malware detection. |
| pyfiglet | This python library is used for the creation of ASCII art fonts. |
| urllib | This python library is used to fetch URLs. |
| pathlib | This python library is used to represent filesystem paths for various operating systems. |
| selenium | This python library is used to retrieve the wasm files from the specified URL. |
| wasm | This python library is used to decode and disassemble WebAssembly modules and bytecode. |
| time | This python library is used to delay the time for the network request. |

## *C. Implementation*

In this section, the technical implementation of the solution will be discussed in detail.

*a)* *URL WebAssembly Scraping:* WebAssembly scraping is performed using the combination of Google Chrome and Selenium Wire. A directory named "Temp" will first be created for the storage of the extracted wasm files. This is followed by the setting of options for Google Chrome. The headless option will be set to run the Chrome browser in a headless environment i.e., without a visible UI shell. On the other hand, the logging option will be disabled to prevent the printing of unnecessary log statements.

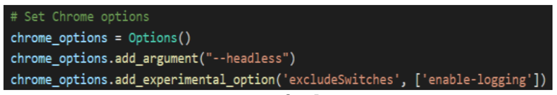
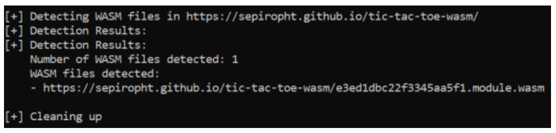

```
# Set Chrome options
chrome_options = Options()
chrome_options.add_argument("--headless")
chrome_options.add_experimental_option('excludeSwitches', ['enable-logging'])
```

Figure 4. Setting of Chrome Options

Next, an instance of a Chrome driver will be created with the options set earlier using the WebDriver library in Selenium Wire. Instead of using a ChromeDriver executable, the WebDriver_Manager library will be used to automatically detect the version of Chrome installed on the user's machine to accurately launch the correct WebDriver version (first downloading the specified executable if required). The WebDriver_Manager library has a caching feature to avoid re-downloading executables that have been downloaded previously [9].

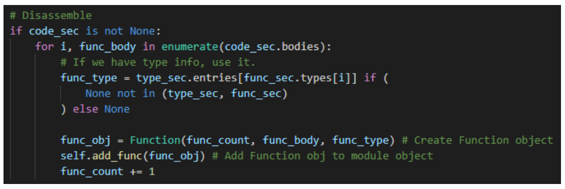
After creating the instance of the Chrome driver, it will first navigate to the specified URL provided by the user. The driver will then capture all network requests made by the website for 5 seconds. Filtering of the network requests to retrieve the full path of all wasm binary files will be subsequently performed by checking that the value of the 'Content-Type' field in the respective network response matches 'application/wasm'. All detected wasm binary files will then be downloaded into the Temp directory using the urllib library. The number of wasm files extracted and the full path to each wasm file will be displayed to the user at the end.



```
[+] Detecting WASM files in https://sepiropht.github.io/tic-tac-toe-wasm/
[+] Detection Results:
[+] Detection Results:
    Number of WASM files detected: 1
    WASM files detected:
    - https://sepiropht.github.io/tic-tac-toe-wasm/e3ed1dbc22f3345aa5f1.module.wasm

[+] Cleaning up
```

Figure 5. Sample Output of Wasm Scraping

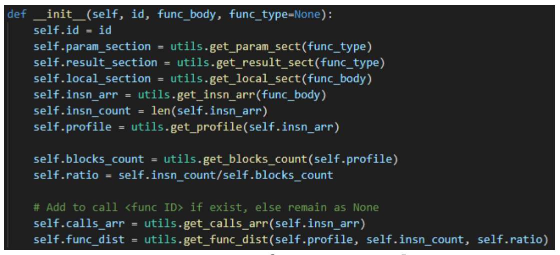
b) *Disassembler:* The python wasm library [5] was utilized to assist in the disassembling of the wasm binary. A Module object will be instantiated and the disassemble function will be called. The function's ID, body, and type are extracted using the library. The total instructions of each function are tabulated and passed as a Function object constructor argument along with the function's body and type. The Function object is then added into an array as shown below.



```python
# Disassemble
if code_sec is not None:
    for i, func_body in enumerate(code_sec.bodies):
        # If we have type info, use it.
        func_type = type_sec.entries[func_sec.types[i]] if (
            None not in (type_sec, func_sec)
        ) else None

        func_obj = Function(func_count, func_body, func_type) # Create Function object
        self.add_func(func_obj) # Add Function obj to module object
        func_count += 1
```

Figure 6. Disassembly, Function Objects

c) *Semantic Profiling:* During the instantiation of each Function object, the function body will be broken down into several attributes of the object. These attributes include the parameter, result, and local section, instructions, instructions' count, function's ID, profile, block count, ratio, inline function calls, and the function distribution. This is shown below in Figure 7.

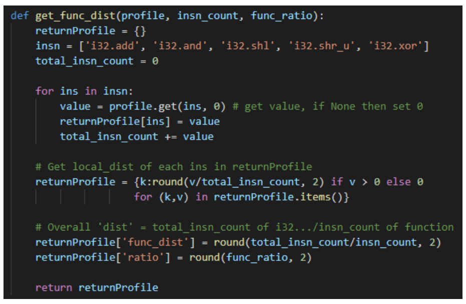


```python
def __init__(self, id, func_body, func_type=None):
    self.id = id
    self.param_section = utils.get_param_sect(func_type)
    self.result_section = utils.get_result_sect(func_type)
    self.local_section = utils.get_local_sect(func_body)
    self.insn_arr = utils.get_insn_arr(func_body)
    self.insn_count = len(self.insn_arr)
    self.profile = utils.get_profile(self.insn_arr)

    self.blocks_count = utils.get_blocks_count(self.profile)
    self.ratio = self.insn_count/self.blocks_count

    # Add to call <func ID> if exist, else remain as None
    self.calls_arr = utils.get_calls_arr(self.insn_arr)
    self.func_dist = utils.get_func_dist(self.profile, self.insn_count, self.ratio)
```

Figure 7. Init of Function Object

The profile of the function will store the count of each unique instruction. The block count will contain the number of specified control instructions [6], in this case, the block, loop, if, else, br, br_if, br_table, return, call_indirect instructions. The ratio will be the total number of instructions divided by the block count. This ratio is a metric that can be used to detect potential malicious functions. Cryptonight for example has instructions that perform multiple numeric instructions [6] in very few blocks which results in a high ratio output.

The function distribution will tabulate the distribution of specified instructions [5], in this case, the i32.add, i32.and, i32.shl, i32.shr_u, i32.xor instructions.



```python
def get_func_dist(profile, insn_count, func_ratio):
    returnProfile = {}
    insn = ['i32.add', 'i32.and', 'i32.shl', 'i32.shr_u', 'i32.xor']
    total_insn_count = 0

    for ins in insn:
        value = profile.get(ins, 0) # get value, if None then set 0
        returnProfile[ins] = value
        total_insn_count += value

    # Get local_dist of each ins in returnProfile
    returnProfile = {k:round(v/total_insn_count, 2) if v > 0 else 0
                    for (k,v) in returnProfile.items()}

    # Overall 'dist' = total_insn_count of i32.../insn_count of function
    returnProfile['func_dist'] = round(total_insn_count/insn_count, 2)
    returnProfile['ratio'] = round(func_ratio, 2)

    return returnProfile
```

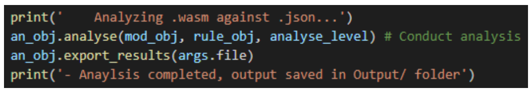
Figure 8. Function Distribution Implementation

There will be 2 distributions tabulated. The first is an overall distribution of the total number of i32.add, i32.and, i32.shl, i32.shr_u, i32.xor instructions divided by the total number of instructions in the function. The second distribution will be the ratio of each of the i32.add, i32.and, i32.shl, i32.shr_u, i32.xor instructions of the total sum of the number of i32.add, i32.and, i32.shl, i32.shr_u, i32.xor instructions.

After each Function object has been initiated and added to the function object array of the Module object, the profile_module method of the Module object will be called. The top 5 Function objects with the highest function distribution will be added to a dictionary.

d) *Rule Generation:* The Module object has a method called generate_rule. This method will write out a JSON rule file based on the top 5 Functions with the highest function distribution.

e) *Analysis:* An Analysis object will be created and the analyze method will be called which takes in a Module, Rule object, and analyse_level which is an integer.



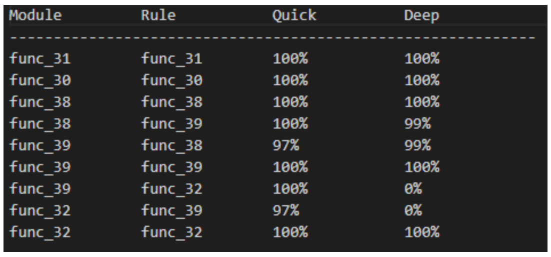
```python
print('    Analyzing .wasm against .json...')
an_obj.analyse(mod_obj, rule_obj, analyse_level) # Conduct analysis
an_obj.export_results(args.file)
print('- Anaylsis completed, output saved in Output/ folder')
```

Figure 9. Analyzing Module and Rule

If the analyse_level is 1, a quick analysis will be performed. If it is 2, then a deep analysis will be performed instead. The quick analysis involves the comparison of each functions' overall function distribution ratio in the Module object against the Rule object's functions' overall distribution ratio. The top 5 functions with the highest amount of similarity will be reported in the analysis output.

If the analyse_level is 2, a deep analysis will be performed. The deep analysis also involves the quick analysis phase but includes a deeper analysis of the function's i32.add, i32.and, i32.shl, i32.shr_u, i32.xor instruction ratio against the rule respective function's i32.add, i32.and, i32.shl, i32.shr_u, i32.xor instruction ratio.

Below is the deep analysis result of the cryptonight.wasm binary against the rule that was generated via the cryptonight.wasm binary. The percentage shows the similarity between the Module's function and the Rule's function.



```
Module          Rule            Quick           Deep
-------------------------------------------------------
func_31         func_31         100%            100%
func_30         func_30         100%            100%
func_38         func_38         100%            100%
func_38         func_39         100%            99%
func_39         func_38         97%             99%
func_39         func_39         100%            100%
func_39         func_32         100%            0%
func_32         func_39         97%             0%
func_32         func_32         100%            100%
```

Figure 10. Deep Analysis Result

f) *Yara Malware Detection:* Yara malware detection program is integrated into Malwasm for signature-based analysis. The usage of the yara python library allows the compilation of rules and verification if the provided yara rules match the specified wasm files. Malwasm provides a modular approach to inputting personalized yara rules. Users can create custom yara rules and move the yar file to the resources/yara_rules folder and execute the Malwasm command. Malwasm will automatically compile all the yara rule files present in that directory for malware detection.

g) *Graph Generation:* Malwasm has the capabilities to generate 3 types of graphs for analysis. Malwasm utilizes wasp.exe, a C++ library that enables the ease of usage working with wasm modules, to generate the call, control flow, and data flow graph. The call graph enables the user to better understand the execution of the program as it depicts the relationship of subroutines in a program. Meanwhile, the control flow graph is more specific whereby it dives into a particular subroutine, mapping out the various branches. Lastly, the data flow graph shows the flow of data within a subroutine without conditional branches. These 3 graphs will provide valuable information regarding malware analysis on the wasm file.

## IV. Comparison With Existing Solutions

At present, there are very limited open-source tools that focus on malicious detection and analysis of wasm. Therefore, the comparisons are only made with reference to an existing tool called Octopus. As the future progresses, there might be more such tools in the future. Table II below states the features present in the proposed solution, Malwasm, and Octopus.

TABLE II
Comparison Between Malwasm And Octopus

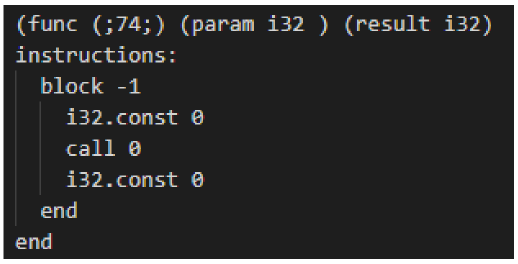
| **Tool Name/Functions** | **Malwasm** | **Octopus** |
|---|---|---|
| Scraping wasm files from URL | ✓ | x |
| Disassembler | ✓ | ✓ |
| Control Graph Analysis | ✓ | ✓ |
| Call Graph Analysis | ✓ | ✓ |
| Data Flow Graph Analysis | ✓ | x |
| YARA detection | ✓ | x |
| Semantic Profiling | ✓ | x |
| Rule Generation | ✓ | x |

## V. Results Analysis

This section will showcase the validity of the proposed solution, Malwasm, and also compare the results against existing open-source solutions in the open market. The open-source tool that the team will be comparing against is Octopus.

### A. Malwasm

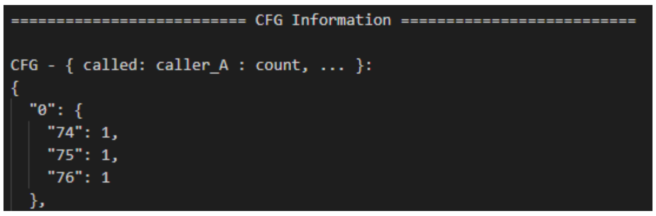
The proposed solution can disassemble the wasm binary and output a pseudo wat file as shown below in Figure 11 and disassembled module information in Figures 12 to 14.



```
(func (;74;) (param i32 ) (result i32)
instructions:
  block -1
    i32.const 0
    call 0
    i32.const 0
  end
end
```

Figure 11. Pseudo Wat File

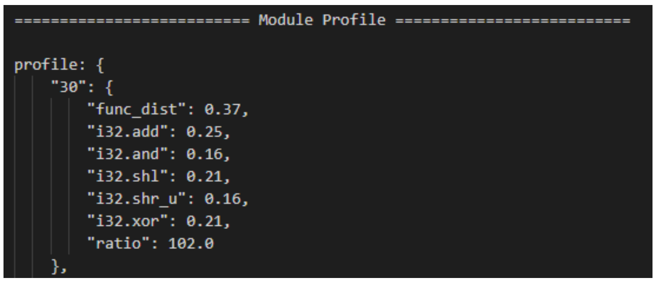
In Figure 12 below, the CFG information shows a JSON format of the CFG information of the module. Each key represents the function's ID, and the associated value is a JSON where each key is the function's ID that calls the outer function ID, and the value represents the number of times the call occurs.



```
========================= CFG Information =========================

CFG - { called: caller_A : count, ... }:
{
  "0": {
    "74": 1,
    "75": 1,
    "76": 1
  },
```

Figure 12. CFG Information Excerpt

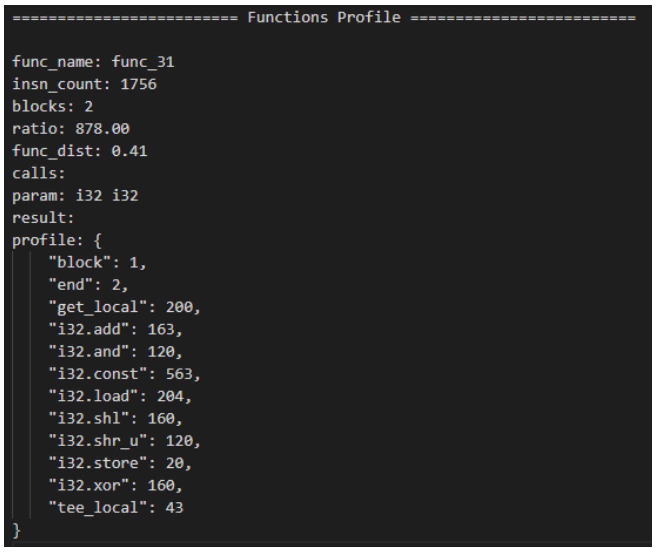
In Figure 13 below, the module profile shows a JSON where the key is the function ID, and the nested JSON shows the function distribution of specified instructions along with the ratio. These are the same statistics mentioned in Section C, subsection c, semantic profiling.



```
========================= Module Profile =========================

profile: {
    "30": {
        "func_dist": 0.37,
        "i32.add": 0.25,
        "i32.and": 0.16,
        "i32.shl": 0.21,
        "i32.shr_u": 0.16,
        "i32.xor": 0.21,
        "ratio": 102.0
    },
```

Figure 13. Module Profile Excerpt

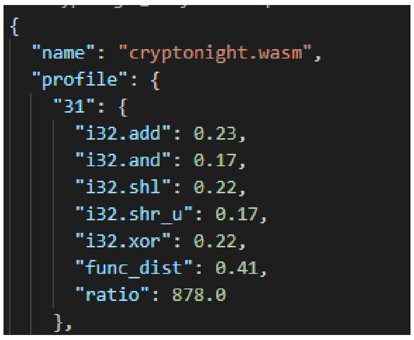
In Figure 14 below, each function of the disassembled wasm binary is profiled and output. The function name, instruction count, block count, ratio, function distribution, function ID calls, parameter, result, section, and the profile is output. The profile tabulates the number of times an instruction is being called.



```
======================= Functions Profile =======================

func_name: func_31
insn_count: 1756
blocks: 2
ratio: 878.00
func_dist: 0.41
calls:
param: i32 i32
result:
profile: {
    "block": 1,
    "end": 2,
    "get_local": 200,
    "i32.add": 163,
    "i32.and": 120,
    "i32.const": 563,
    "i32.load": 204,
    "i32.shl": 160,
    "i32.shr_u": 120,
    "i32.store": 20,
    "i32.xor": 160,
    "tee_local": 43
}
```

Figure 14. Function profile excerpt

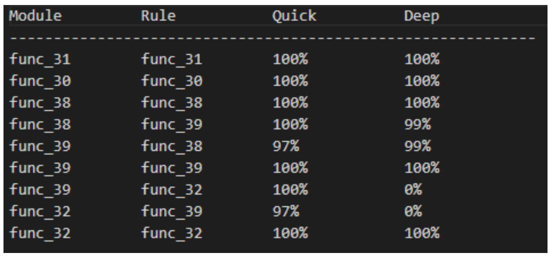
It will also output a JSON rule of the profiled wasm that can be used during analysis. The JSON rule is shown below in Figure 15.



```
{
  "name": "cryptonight.wasm",
  "profile": {
    "31": {
      "i32.add": 0.23,
      "i32.and": 0.17,
      "i32.shl": 0.22,
      "i32.shr_u": 0.17,
      "i32.xor": 0.22,
      "func_dist": 0.41,
      "ratio": 878.0
    },
```

Figure 15. JSON Rule Excerpt

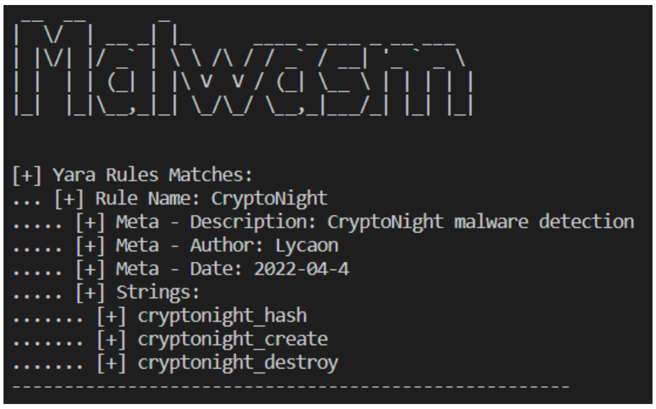
It is also able to detect the functions in cryptonight.wasm not based on the string detection but semantic profiling of instructions as shown below in Figure 16.



```
Module          Rule            Quick           Deep
------------------------------------------------------------
func_31         func_31         100%            100%
func_30         func_30         100%            100%
func_38         func_38         100%            100%
func_38         func_39         100%            99%
func_39         func_38         97%             99%
func_39         func_39         100%            100%
func_39         func_32         100%            0%
func_32         func_39         97%             0%
func_32         func_32         100%            100%
```

Figure 16. Malwasm Deep Analysis Report

Malwasm has the capabilities to use yara rules on the specified wasm file for the detection of malware. A screenshot of the yara results on a cryptonight wasm file is shown in Figure 17 below.



```
 __  __       _                               
|  \/  | __ _| |_      ____ _ ___ _ __ ___  
| |\/| |/ _` | \ \ /\ / / _` / __| '_ ` _ \ 
| |  | | (_| | |\ V  V / (_| \__ \ | | | | |
|_|  |_|\__,_|_| \_/\_/ \__,_|___/_| |_| |_|


[+] Yara Rules Matches:
... [+] Rule Name: CryptoNight
..... [+] Meta - Description: CryptoNight malware detection
..... [+] Meta - Author: Lycaon
..... [+] Meta - Date: 2022-04-4
..... [+] Strings:
....... [+] cryptonight_hash
....... [+] cryptonight_create
....... [+] cryptonight_destroy
--------------------------------------------------
```

Figure 17. Yara Results

In addition, Malwasm can also act as an analysis platform for users to analyze the wasm file with the generation of the call graph, control flow graph, and data flow graph.

The call graph of cryptonight is as shown in Figure 18 below.

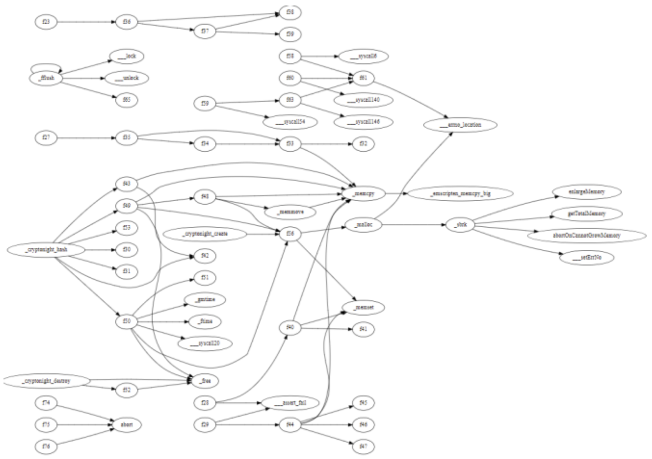


Figure 18. Call Graph of CryptoNight

The control flow graph of the _cryptonight_create function in cryptonight is as shown in Figure 19 below.

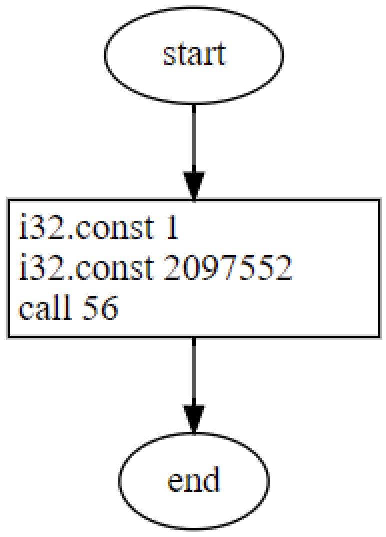


Figure 19. Control Flow Graph of _cryptonight_create function from CryptoNight

The data flow graph of the _cryptonight_create function in cryptonight is as shown in Figure 20 below.

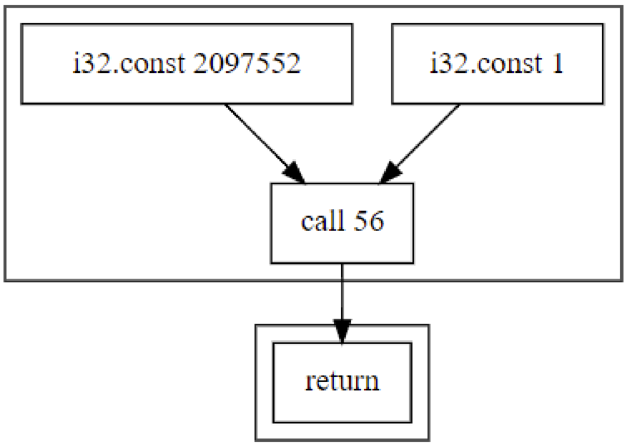


Figure 20. Data Flow Graph of _cryptonight_create function from CryptoNight

## *B. Octopus*

The Octopus security analysis framework is an existing tool to perform behavioral analysis on Web Assembly files. The tool allows multiple visualizations of data to break down the WebAssembly files such as visualizing the instruction count for the functions as shown in Figure 21 below.

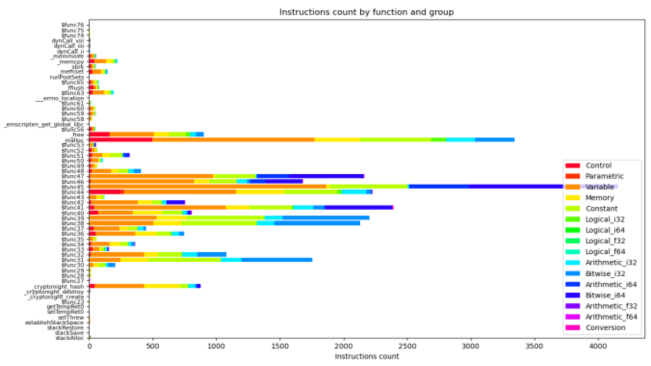


Figure 21. Instructions Count of CryptoNight

Octopus has the capability to generate a call graph to perform static analysis on the wasm file, and view what are the subroutines or functions that are called in relation to each other. A snippet of the _cryptonight_hash function in cryptonight is shown in Figure 22 below.

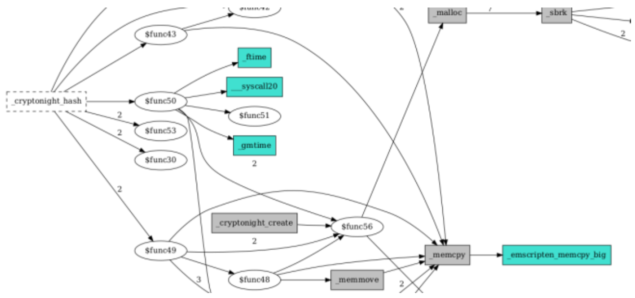


Figure 22. Snippet of the Call Graph of CryptoNight

There are flaws to the program such as control flow analysis, it is too complex to view and determine the flow of the WASM program. An example shown below is what it looks like when generating a control flow graph for CryptoNight.

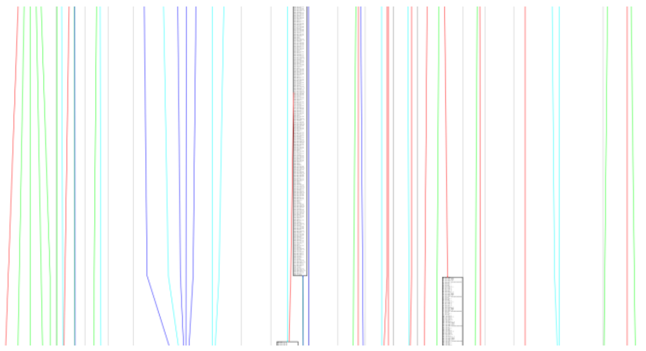
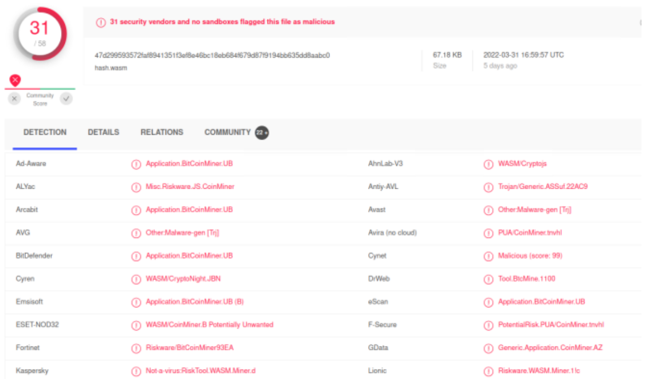


Figure 23. Control Flow Analysis of CryptoNight

## *C. VirusTotal*

There are 31 security vendors flagging and identifying the cryptonight wasm file as a malicious file as shown in Figure 24 below. It seems to be a concern that not every vendor is able to identify the malicious WebAssembly file.



31 / 58

31 security vendors and no sandboxes flagged this file as malicious

47d299593572faf8941351f3ef8e46bc18eb684f679d87f9194bb635dd8aabc0
hash.wasm

67.18 KB Size

2022-03-31 16:59:57 UTC 5 days ago

Community Score

DETECTION DETAILS RELATIONS COMMUNITY 22

| | | | |
|---|---|---|---|
| Ad-Aware | Application.BitCoinMiner.UB | AhnLab-V3 | WASM/Cryptojs |
| ALYac | Misc.Riskware.JS.CoinMiner | Antiy-AVL | Trojan/Generic.ASSuf.22AC9 |
| Arcabit | Application.BitCoinMiner.UB | Avast | Other:Malware-gen [Trj] |
| AVG | Other:Malware-gen [Trj] | Avira (no cloud) | PUA/CoinMiner.tnvhl |
| BitDefender | Application.BitCoinMiner.UB | Cynet | Malicious (score: 99) |
| Cyren | WASM/CryptoNight.JBN | DrWeb | Tool.BtcMine.1100 |
| Emsisoft | Application.BitCoinMiner.UB (B) | eScan | Application.BitCoinMiner.UB |
| ESET-NOD32 | WASM/CoinMiner.B Potentially Unwanted | F-Secure | PotentialRisk.PUA/CoinMiner.tnvhl |
| Fortinet | Riskware/BitCoinMiner93EA | GData | Generic.Application.CoinMiner.AZ |
| Kaspersky | Not-a-virus:RiskTool.WASM.Miner.d | Lionic | Riskware.WASM.Miner.1!c |

Figure 24. VirusTotal Scan of CryptoNight

## VI. Conclusion

Adversaries are looking in the direction of WebAssembly for attack vectors as it gains traction. In this paper, the team has introduced the tool Malwasm to detect possible malicious behavior in WebAssembly files. It will require much more research to elevate Malwasm to be capable of detecting complex malicious WebAssembly files. In future work, to detect WebAssembly files that are heavily obfuscated or to address new patterns that arise in identifying malicious behavior. There is the possibility of utilizing machine learning to identify such behavior and to transgress into a much greater tool that could possibly detect polymorphic variants of malware.

## REFERENCES

[1] J. Ruth, T. Zimmermann, K. Wolsing, and O. Hohlfeld, "Digging into browser-based crypto mining," 2018. [Online]. Available: https://arxiv.org/pdf/1808.00811.pdf [Accessed 5 April 2022].

[2] Tahir, R., "A Study on Malware and Malware Detection Techniques," 2018. [Online] Mecs-press.org. Available at: https://www.mecs-press.org/ijeme/ijeme-v8-n2/IJEME-V8-N2-3.pdf [Accessed 5 April 2022].

[3] A. Haas, A. Rossberg, D. L. Schuff, B. L. Titzer, M. Holman, D. Gohman, L. Wagner, A. Zakai, and J. Bastien, "Bringing the Web up to Speed with WebAssembly," 2017. [Online]. Available: https://dl.acm.org/doi/pdf/10.1145/3062341.3062363 [Accessed 5 April 2022].

[4] W. Wang, B. Ferrell, X. Xu, W. Kevin, Hamlen, and S. Hao, "SEISMIC: SEcure In-lined Script Monitors for Interrupting Cryptojacks," 2018. [Online]. Available: https://www.researchgate.net/publication/326857606_SEISMIC_SEcure_In-lined_Script_Monitors_for_Interrupting_Cryptojacks [Accessed 5 April 2022].

[5] J. Höner, "GitHub - athre0z/wasm: WebAssembly decoder & disassembler library", GitHub. [Online]. Available: https://github.com/athre0z/wasm. [Accessed: 05 April 2022].

[6] "Instructions — WebAssembly 1.1 (Draft 2022-04-05)", Webassembly.github.io. [Online]. Available: https://webassembly.github.io/spec/core/syntax/instructions.html#. [Accessed: 05 April 2022].

[7] W3C WebAssembly Community Group. WebAssembly design documents. [Online]. Available: https://webassembly.org [Accessed: 05 April 2022].

[8] M. Mihai, "WebAssembly Is Abused by eCriminals to Hide Malware," 2021. [Online]. Available: https://www.crowdstrike.com/blog/ecriminals-increasingly-use-webassembly-to-hide-malware/ [Accessed: 05 April 2022].

[9] G. Boni, "WebDriverManager," 2022. [Online]. Available: https://bonigarcia.dev/webdrivermanager/ [Accessed: 05 April 2022].

[10] N. F. Faraz, "A Deep-Learning Based Robust Framework Against Adversarial P.E. and Cryptojacking Malware," 2020. [Online]. Available: https://digitalcommons.fiu.edu/cgi/viewcontent.cgi?article=5788&context=etd [Accessed 05 April 2022].